COPY

(2)

AD-A220 790

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE — READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NW-LIS-89-12-02
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): Wirelisp Manual
5. TYPE OF REPORT & PERIOD COVERED: Technical
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Carl Ebeling, Zhanbing Wu
8. CONTRACT OR GRANT NUMBER(s): N00014-88-K-0453
9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Northwest Laboratory for Integrated Systems
University of Washington
Dept. of Comp. Science, FR-35 Seattle, WA 98195
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS:
DARPA-ISTO
1400 Wilson Boulevard
Arlington, VA 22209
12. REPORT DATE: December 1989
13. NUMBER OF PAGES: 21
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
Office of Naval Research - ONR
Information Systems Program - Code 1513: CAF
800 North Quincy Street
Arlington, VA 22217
15. SECURITY CLASS. (of this report): Unclassifed
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT (of this Report):
Distribution of this report is unlimited.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:
19. KEY WORDS (Continue on reverse side if necessary and identify by block number):
Lisp, TLisp, Graphical description, Procedural description, VLSI circuits, schematic, parametrized schematic.
20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

WireLisp is a hierarchical circuit structure description language that combines the intuition of schematics and the generality of procedural description. It is different from other similar tools in that the schematics and procedural descriptions are closely intertwined. More specifically, WireLisp is embedded in Lisp but provides graphical constructs for the most common procedural constructs. A WireLisp program consists of a set of device definitions, each described in the most convenient way:

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 Abstract (continued from front page)

Lisp expressions may be embedded in schematics and schematics may be embedded in Lisp as well. This allows descriptions to be highly expressive, yet easily specified and understood. This manual defines the WireLisp language.

WireLisp Manual

Zhanbing Wu and Carl Ebeling

Technical Report #89-12-02

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA

WireLisp Manual

Zhanbing Wu   Carl Ebeling

Contents

1 Getting Started

1.1 What is WireLisp?

WireLisp is a hierarchical circuit structure description language that combines the intuition of schematics and the generality of procedural description. It is different from other similar tools in that the schematics and procedural descriptions are closely intertwined. More specifically, WireLisp is embedded in Lisp but provides graphical constructs for the most common procedural constructs. A WireLisp program consists of a set of device definitions, each described in the most convenient way: Lisp expressions may be embedded in schematics and schematics may be embedded in Lisp as well. This allows descriptions to be highly expressive, yet easily specified and understood.

This manual defines the WireLisp language. Wirelisp is implemented in Lisp and arbitrary Lisp can be used in Wirelisp programs. A separate manual, *Drawing WireLisp*, describes how to draw WireLisp constructs.

1.2 Running WireLisp

The current implementation of WireLisp uses the **Xdp** drawing program as the graphics front end and **T** as the base Lisp language. **Xdp** is an interactive drawing editor developed at Carnegie-Mellon University, **T** is a dialect of Scheme developed at Yale University.

1.2.1 Setting Up

On VLSI machines, you should:

- make sure you are using X11 Version 3 window system;
- make sure **/usr/bin/X11**, **/u2/t/bin** and **$UW_VLSI_TOOLS/bin** are on your search path;
- set up the environment variable **DPPATH** which contains the search paths for your **Xdp** files (with **.dp** extension);
- set the environment variable **WLPATH** which contains the search paths for your WireLisp files (with **.wl** extension); [1]
- set up the **T** initialization file **init.t** at your *home directory*. This file is automatically loaded upon entering **T**.

1.2.2 A Sample Session

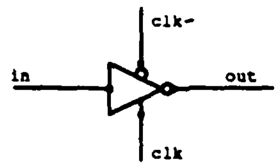
In this section, we will go through a simple example to show how WireLisp programs are executed.

Suppose we want to create the **.sim** file for a device named `clkinv` and we use the default CMOS library which includes the primitives `ptrans` and `etrans`. There are four basic steps:

1. Use **Xdp** to prepare the drawing for `clkinv` (see Figure 1) and save it in the file **clkinv.dp**;
2. Enter T (version 3.1). Note that the default initialization file is automatically loaded which in turn requests the loading of the Wirelisp initialization file.
3. Load the WireLisp interpreter by typing `(wirelisp)`. By default, the device library is CMOSLIB and the output file extension is **.sim**.
4. Make the root device, which in this case is `clkinv` by typing `(make clkinv)`.

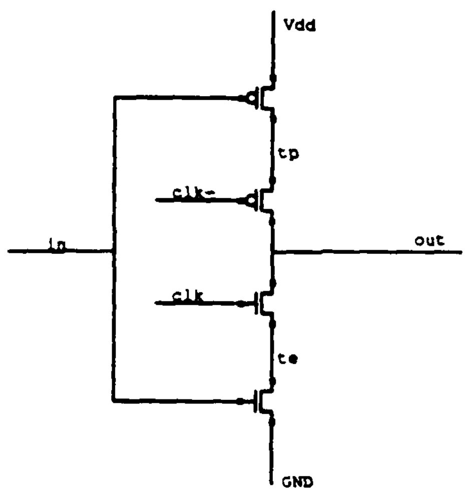
[1] This is not implemented yet.

clkinv



(access in out clk clk-)

(local tp te)



UW/LIS

Figure 1: *the circuit drawing for* `clkinv`

```
stehekin% t3.1
T 3.1 (5) MC68000/UNIX Copyright (C) 1988 Yale University
;Loading /u1/zhanbing/init.t into USER-ENV
; Loading /usr/lis/src/vlsi/vlsi-3.2/lib/wirelisp/wirelisp.mo into USER-ENV
T Top level
) (wirelisp)
;Loading /usr/lis/src/vlsi/vlsi-3.2/lib/wirelisp/device.mo into USER-ENV
;Loading /usr/lis/src/vlsi/vlsi-3.2/lib/wirelisp/instance.mo into USER-ENV
;Loading /usr/lis/src/vlsi/vlsi-3.2/lib/wirelisp/signal.mo into USER-ENV
 ...
;Loading /usr/lis/src/vlsi/vlsi-3.2/lib/wirelisp/cmoslib.mo into USER-ENV
) (make clkinv)
Running dp2wl on clkinv.dp ...
pre-process clkinv.wl ...
;Loading clkinv.wlpp into USER-ENV
;no value
) (exit)
stehekin% more clkinv.wl
 (-device- (clkinv out in clk clk-)
      (access in out clk clk-)
      (local tp te)
      (-I- ptrans in Vdd tp)
      (-I- ptrans clk- tp out)
      (-I- etrans clk out te)
      (-I- etrans in te GND))
stehekin% more clkinv.sim
| units:  100.0 tech:  cmos-s format:  UCB
p CLK- VDD TP 2 2
p IN TP OUT 2 2
e IN OUT TE 2 2
e CLK TE GND 2 2
stehekin%
```

clkinv.wl is derived from **clkinv.dp** by running the schematics extractor **dp2wl**, and **clkinv.wlpp**, which is deleted after loading, is derived from **clkinv.wl** by running a preprocessor. The output file is named after the root device, which in this case is **clkinv.sim**.

Note that in the above sample session, we used the standard **T** heap size, i.e. 4096000 bytes. However, the user may need to use a smaller heap size due to inadequate memory. The heap size is specified with the -h flag. For example,

```
% t3.1 -h 1024000
```

claims a heap of 1024000 bytes. If the given size is smaller than the minimum required heap size (524280 bytes), **T** will use the minimum size instead.

2 Introduction

In this section, we will go through the design of `clkinv` to show some basic concepts of WireLisp. Since this manual focuses on the procedural aspect, we will examine the **.wl** files. [2]

In WireLisp, each device (or module) is a procedure. A device is instantiated by a call on this procedure. Because a device procedure may have parameters, it actually defines a family of devices, instances of which

[2] In practice, the user makes the drawings and the system automatically creates the .wl files by running **dp2wl**.

are produced by invoking the procedure with particular parameter assignments. For example, let us look the device procedure of clkinv.[3]

```
;; 'clkinv' has 4 signal parameters:  out, in, clk, clk-.
(-device- (clkinv out in clk clk-)
     ;; all 4 signal parameters are accessed as simple signals.
     ;; This is required only if clkinv is the root device.
     (access out in clk clk-)
     ;; there are two local signals which are visible only inside 'clkinv'.
     (local tp te)
     ;; 'clkinv' is comprised of the leaf devices --- P and E transistors.
     ;; 'ptrans' is instantiated with the default width (2) and length (2).
     (-I- ptrans in Vdd tp)
     ;; Connections that share the same signal are physically wired together.
     ;; Ex.drain of one ptrans is wired to source of another via sharing 'tp'.
     (-I- ptrans clk- tp out)
     ;; 'etrans' is instantiated with the default width (2) and length (2).
     (-I- etrans in GND te)
     (-I- etrans clk te out))
```

An instance of clkinv is formed by instantiating ptrans and etrans in turn and connecting the resulting four instances together. Device instances are connected by signal names. When a device is instantiated, the signal names are passed as parameters to the device procedure. All signals must be declared before they are used. There are three types of signals in a device definition:

- *Global signals* are visible to all device instances. WireLisp provides two default global signals: Vdd and GND.
- *Local signals* are visible only within the device in which they are declared. For example, clkinv has two local signals: tp and te.
- *Signal parameters* are passed in from other devices. For example, clkinv has four signal parameters: out, in, clk and clk-.

Signals can be defined to have structures, that is, composed hierarchically of other signals. These signals are called *cables.* Both homogeneous structures (BUS) and heterogeneous structures (SN) can be specified. The use of structured signals increases the level of abstraction and decreases the size of the program by bundling several related signals into one cable. For example, clkinv can be defined equivalently as:

```
;; 'clkinv' has 3 signal parameters:  'out', 'in', 'phi'.
(-device- (clkinv out in phi)
     ;; 'out' and 'in' are simple signals, but 'phi' has a structure.
     (access out in (phi (SN H L)))
     (local tp te)
     (-I- ptrans in Vdd tp)
     (-I- ptrans phi.L tp out)           ; 'clk-' is replaced by 'phi.L'.
     (-I- etrans in GND te)
     (-I- etrans phi.H te out))          ; 'clk' is replaced by 'phi.H'.
```

The -I- function for instantiating a device can be intermixed freely with other Lisp code. For example, we can expand the original clkinv to N bits:

[3] Comments are preceded by one or more semicolons and ended by the end of the line.

```
(-device- (N-clkinv out in phi N)
     (access (out in (BUS 1 N)))          ; N input/output bits.
     (access (phi (SN H L)))               ; a common clock signal.
     (repeat i 1 N
          (-I- clkinv out[i] in[i] phi)))
```

This defines a 3-level device hierarchy: **N-clkinv** → **clkinv** → **ptrans** and **etrans**. When a hierarchical circuit description is fully elaborated, all that remains is the entire set of interconnected leaf devices. If each instance of the leaf devices simply outputs a line describing itself, then these outputs are concatenated to form the output of the program. For example, the leaf devices in the default CMOS library are **ptrans** and **etrans**. If we want to create the net list for **N-clkinv** with N = 2, we execute:

(make N-clkinv 2)

The output, saved in the default file **N-clkinv_2.sim**, [4] would be:

```
| units:  100.0 tech:  cmos-s format:  UCB
p IN[1] Vdd CLKINV-1/TP 2 2
p PHI.L CLKINV-1/TP OUT[1] 2 2
e IN[1] GND CLKINV-1/TE 2 2
e PHI.H CLKINV-1/TE OUT[1] 2 2
p IN[2] Vdd CLKINV-2/TP 2 2
p PHI.L CLKINV-2/TP OUT[2] 2 2
e IN[2] GND CLKINV-2/TE 2 2
e PHI.H CLKINV-2/TE OUT[2] 2 2
```

Note that the names of local signals in **clkinv**, **tp** and **te**, have been prepended with the distinct instantiation paths: **clkinv-1/** and **clkinv-2/**. This is necessary in order to distinguish their occurrences in different instances of **clkinv**.

In the next few sections, we will formally define the syntax and semantics of WireLisp constructs. We start with *devices* and then discuss *signals*.

3 Device Definitions

A device is a procedure that defines a family of circuits with similar structure. The parameters of the device determine which instance of this family is to be built. A device is defined by:

(-device- (*dev-name* *signal*$_1$ *signal*$_2$ ... *signal*$_m$
gen-param$_1$ *gen-param*$_2$ ... *gen-param*$_n$
((*opt-param-name*$_1$ *default-value*$_1$) ... (*opt-param-name*$_k$ *default-value*$_k$)))
access-declarations
local-declarations
temporary-variable-bindings
device-description)

where,

1. *dev-name* is the name of the defined device. The keyword **-device-** is parallel to the keyword **define** in **T**.

2. *signal*$_1$ *signal*$_2$ ... *signal*$_m$ are formal signal parameters. Their actual values determine how the instantiated device is connected with other parts of the circuit. Signal parameters are mandatory and

[4] The parameter value 2 is appended to the device name in order to distinguish between different instances of the same family.

passed by position, i.e. the first actual signal is passed to the first formal signal.

3. *gen-param*$_1$ *gen-param*$_2$ ... *gen-param*$_n$ are formal general parameters. They can be of any type except signals. Typically they are integers that specify the size of a circuit, such as `N` in the device `N-clkinv`. General parameters are also mandatory and passed by position.

4. *((opt-param-name*$_1$ *default-value*$_1$*) ...)* are optional parameters which are passed by name, just like keyword arguments in Common Lisp[5]. The order of these parameters does not matter. All optional parameters must be declared in the device definition. If a device is instantiated with an undefined optional parameter, WireLisp simply ignores that parameter.

   There is a special optional parameter called **`instance-name`** (or **`IN`** in short) used to specify names of device instances. It can be thought as a system-defined optional parameter, i.e. every device automatically defines it as an optional parameter. Device instance names are used to construct the unique device instantiation path names (see Section 5.2). Normally, the user ***should not assign the same name to different instances within a device definition***. When a device is instantiated without an instance name, the system will automatically generate a unique instance name.

5. *access-declarations* is a sequence of **`access`** declarations which specify how cable parameters are accessed inside *this* device. Note that this declaration is only necessary for those parameters whose components are to be accessed.

6. *local-declarations* is a sequence of **`local`**, **`slocal`** and **`shadow`** declarations which create signals visible only within the device. For cables, this declaration creates not only their wire structure but also the access structure.

7. *temporary-variable-bindings* is a sequence of binding expressions like:
   `(defvar (var`$_1$ `value`$_1$`) ... (var`$_K$ `value`$_K$`))`
   For example,
   `(defvar (N (size-of in)) (N1 (- N 1)))`
   `defvar` resembles very much like **`let*`** in **T**, i.e. the bindings evaluated first are visible to later evaluations. Note that *no* **`defvar`** binding is visible to any signal declaration.

8. *device-description* is a sequence of intermixed T and WireLisp expressions which define how the device is constructed.

4 Device Instantiations

In the object-oriented view, a device definition characterizes the common properties of a class of objects, and a device instantiation creates an object of that class. When a device procedure is called with actual parameters, an instance of that device structure is created. All instances of a device share the same or similar circuit structure, but differ in what actual parameters are used in their instantiations.

4.1 Instantiating a Device

A device can be instantiated via:

`(-I-` *dev-name signal*$_1$ *signal*$_2$. *signal*$_m$
gen-param$_1$ *gen-param*$_2$ ... *gen-param*$_n$
(**prop** (*opt-param-name*$_1$ *value*$_1$) ... (*opt-param-name*$_k$ *value*$_j$)))

where, the number and the order of mandatory parameters, i.e. signals and general parameters, must match the corresponding formal parameters in the definition. The order of optional parameters does not matter because they are passed by name. The keyword **prop** indicates that the following is a list of optional

[5] T does not have keyword arguments.

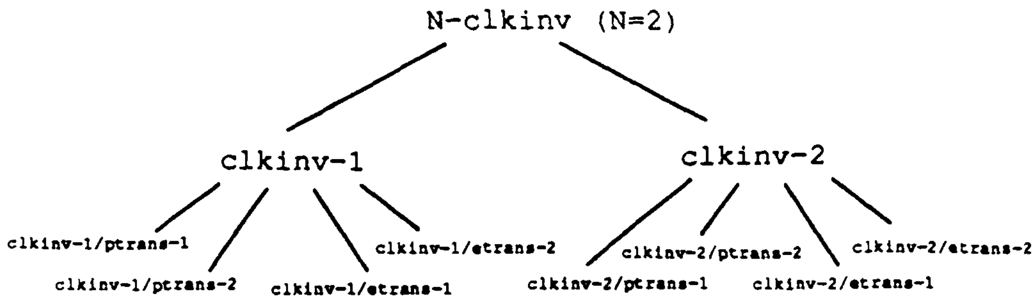


Figure 2: *the instantiation hierarchy in* **(make N-clkinv 2)**

parameter pairs: (*name value*). If there is no actual value specified for an optional parameter, WireLisp fills in its default value as given in the definition of that device. If a value is given for an undefined optional parameter, it will be ignored.

4.2 Hierarchical Instantiation Path Names

A global instance name can be defined by an instantiation path formed by concatenating local instance names along the path of the instantiation from the root. Since local instance names should be unique within a device, the result of the concatenation is also globally unique.

Local instance names may be generated by the system automatically or specified by the user via the **IN** optional parameter. For example, the execution of:

(make N-clkinv 2)

(on page 7) constructs a tree of device instances as shown in Figure 2. These local instance names are generated by the system, e.g. **clkinv-1** and **clkinv-2**.

5 Signals

5.1 Basic Concepts

Devices are connected via signals. A formal signal parameter of a device can be thought as a connection point of that device. When two connection points share the same signal, they are thought to be connected by a wire in the circuit.

There are two types of signals:

1. *Simple signals* represent atomic wires. For example, Vdd and GND are simple signals.
2. *Bundled signals* (or *cables*) represent structured groups of wires. A cable is an ordered list of signals, each of which may be a simple signal or another cable. This recursive definition allows signals to be structured hierarchically thus forming a tree structure. The leaves of the tree are simple signals, and the internal nodes correspond to sub-cables.

A signal is bound to a *physical structure* (or *wire structure*) and an *access structure*. The wire structure represents the actual group of wires corresponding to the signal, while the access structure defines how each individual wire or sub-cable is accessed.

Signals must be declared before use. Once declared, they are used as parameters of devices. Every signal appearing in a device is declared either locally or as a formal parameter. If the formal parameter is a cable, then its access structure must be declared explicitly unless the device has no interest in accessing *individual wires* in the cable. That is, if the device simply wants to pass the formal parameter to other devices it is not necessary to declare an access structure for the parameter.

WireLisp provides two types of structured signals: homogeneous structures (BUS) similar to *arrays* in PASCAL, and the heterogeneous structures (SN) similar to *records* in PASCAL.

5.2 Access Mechanisms: SN and BUS

The components of hierarchical signals, i.e. cables, are referenced via two access mechanisms:

1. *BUS structure* whose components are accessed via an integer index.

   For example,
   `(access (in (BUS 1 N)))`
   declares that the components of `in` can be accessed via:
   `in[1], in[2], ... in[N]`
   respectively. The index may also go from high to low.

2. *Structured Net* (SN) whose components are accessed explicitly by name.

   For example,
   `(access (load (SN H L)))`
   declares that the components of `load` can be accessed via:
   `load.H, load.L`
   respectively.

The complete definition of the access mechanisms, in the form of the extended BNF, is given in the appendix B. Both BUS structures and SN structures can be nested. That is, the component of a BUS structure can be another BUS structure or an SN structure, and the component of an SN structure can be another SN structure or a BUS structure.

Each access declaration effectively constructs a tree of names through which the individual wires are accessed. For example, the access declaration corresponding to the tree in Figure 3 is:

```
(access (sysBus (SN (ctrl (SN RW STROBE ACK)) (addr (BUS 0 23)) (data (BUS 1 32)))))
```

This can be more easily represented in drawing. See *Drawing WireLisp*.

The fundamental difference between an SN and a BUS structure is that the access names in a BUS structure may contain *variable* indices which are evaluated at run-time, while all access names in an SN structure are evaluated before execution starts.

5.3 Signal Declarations: local, slocal, shadow, global, access

The wire structure of a signal represents the physical wires that connect devices. Once created, the signal physical structure never changes. The basic mechanism for creating signals is the local declaration.

5.3.1 Local Declarations

A local declaration does three things:

1. It creates the named signal.

2. It creates the signal access structure which is used in the *current* device;

3. It creates the signal wire structure.

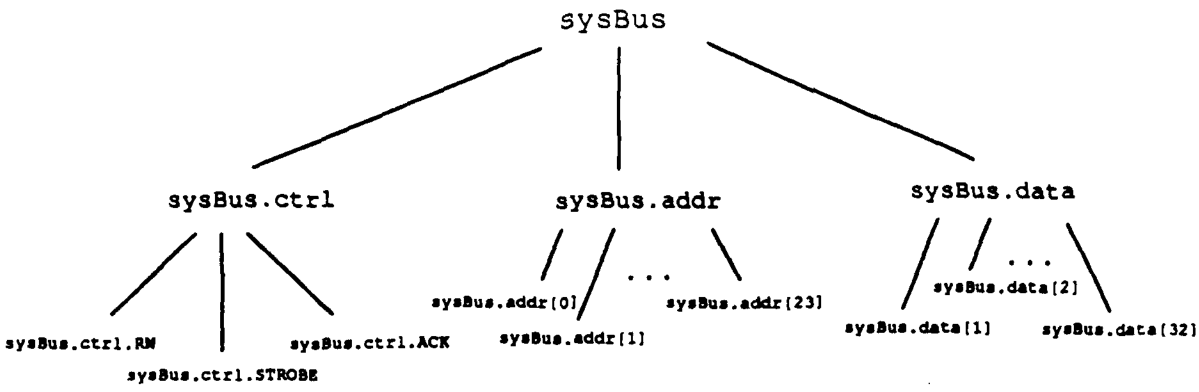


Figure 3: *the access tree* **sysBus**

For example,

`(local (phi (BUS 1 2 (SH H L))))`

creates the signal **phi** and the 4-phase clock comprised of 4 atomic wires:

`phi[1].H, phi[1].L, phi[2].H, phi[2].L (Figure 4 (a)).`

They can be accessed via the structure `(BUS 1 2 (SH H L))` (Figure 4 (b)). For example, the name **phi[1]** accesses the sub-structure `(phi[1].H phi[1].L)`.

The general form of a local declaration is:

(local (*name1 name2 ... nameK access-structure-spec*) **...)**

Each component, (*name1 name2 ... nameK access-structure-spec*), creates a list of named signals that have the same structure. If *access-structure-spec* is missing, simple signals are created instead. In this case, the pair of enclosing parentheses is also eliminated.

Local signals in different instances of a device are differentiated by prepending a unique device instantiation path (discussed in Section 5.2) to their locally declared names (or *local names*). Thus every signal is uniquely identified by a path which locates the device instance.

However, device path names can be very long if the device hierarchy is deeply nested. This may cause storage problems for very large circuits. The user may get round this problem by using *short-named signals* (or *short signals*). They are declared similarly using the reserved word **slocal** instead of **local**. Short signals are named by prepending their local names with a name which is formed by appending a unique integer to the device name. For example, suppose in a device named **foo**, we declare that:

`(slocal (phi (BUS 1 2 (SH H L))))`

and the appending integer happens to be 5 when this declaration is evaluated. Then in the output, the four wires of **phi** will be identified as:

`foo-5/phi[1].H, foo-5/phi[1].L, foo-5/phi[2].H, foo-5/phi[2].L`

Since the user probably does not want to observe all signals, he/she may declare those non-interesting signals as short signals.

By default, the unique integer series starts at 1. But the user may specify a different seed *before executing the program* via:

(set seed *num*) **; the series starts at** *num*.

Whenever a device containing **slocal** declarations is instantiated, the seed is incremented by 1.

Short signals makes debugging difficult because they can be impossible to locate. The user may turn off *all* short signal declarations by:

`(set SHORT 'off)`

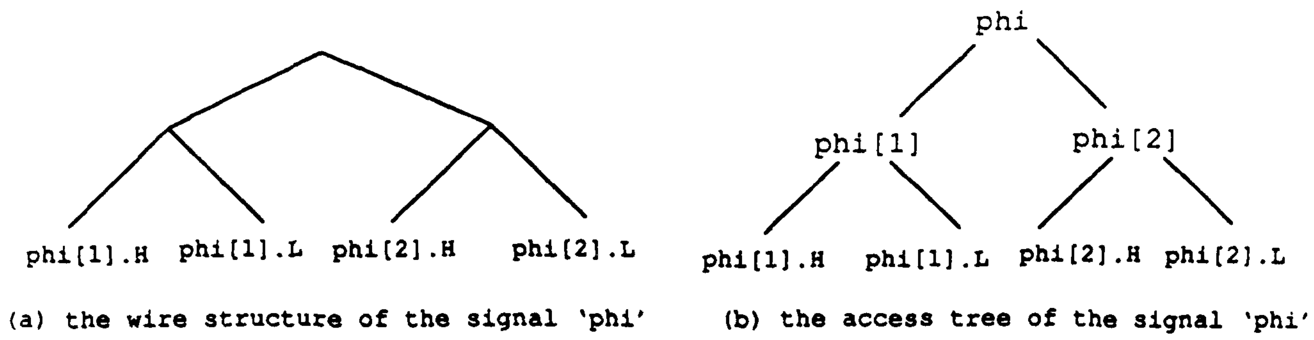


Figure 4: *the wire structure and access structure of* **phi**

which causes WireLisp to treat all **slocal** declarations as **local** declarations. By default, the flag **SHORT** is on.

All signals must be defined before use. Moreover, this definition must be complete. In some cases, however, a device may not understand the structure of the wire connecting two of its constituent devices. Fortunately, the user may get around this problem via **shadow** declarations:

(**shadow** *name1 name2 ... nameK*)

A shadow declaration simply creates a list of *unbound* signals. These unbound signals have neither wire structure nor access structure. These signals remain undefined until some component device provides a declaration. This is described in the next section.

Signals can be declared globally as well. Global signals are visible to all devices from the point of declaration. A global declaration has the same syntax as a local declaration except that the reserved word **local** is replaced by **global**.

WireLisp provides two default global signals: **Vdd** and **GND**.

5.3.2 Access Declarations

When the user wants to access the components of a cable parameter, he/she must specify an access structure for that parameter. The access structure can be anything as long as it matches the wire structure of the actual parameter. This matching is defined as:

Suppose **A** *is the tree corresponding to an access structure, and* **B** *is the tree corresponding to a physical structure.* **A** *matches* **B** *if and only if:*

1. **A** *is a leaf (and therefore,* **A***'s name is used to access the the wire structure rooted at* **B***), or:*

2. *Assume* $A = (a_1\ a_2\ \ldots\ a_m)$ *and* $B = (b_1\ b_2\ \ldots\ b_n)$*. Then,* $m = n$ *and* a_i *matches* b_i*, for all* $i = 1, 2, \ldots m$.

This implies that the physical structure may be deeper than the access structure. That is, the user only needs to specify the access structure down to the level of interest. If the components of a cable are not accessed in a device at all, then there is no need to specify its access structure.

The above definition also implies that a BUS access structure can be used to access a heterogeneous cable as long as the number of indices matches the size of the cable. For example, suppose the wire structure ((a b) c) is passed to the formal parameter foo which is declared as:

```
(access (foo (BUS 1 2)))
```

Then,

```
foo[1] → (a b)
foo[2] → c
```

are all legal. The structure matching is checked at run-time. If the access structure does not match the physical structure, an error is reported and the execution aborts.

There is a special case in declaring BUS **access** structures. That is, the upper bound of the BUS index may be unbound (?), such as:

```
(access (foo (BUS 1 ?)))
```

When WireLisp checks structure matching, the question mark will be replaced by an integer which forces the access structure to match the wire structure. For example, if `((a b) c)` is passed to `foo` above, ? would be replaced by 2. The following is a more complicated example:

```
(-device- (foo1)
      (local (X (SN (H L (BUS 1 8 (SN H L))))))
      (-I- foo2 X)
      ...)
(-device- (foo2 Y)
      (access (Y (SN H L (BUS 1 ?))))
      ...)
```

When the access declaration in `foo2` is evaluated, according to the recursive definition of matching, the question mark will be replaced by 8.

The above discussion assumes that the actual parameter is bound, i.e. declared via **local** or **slocal**. In case the actual parameter is unbound, i.e. declared via **shadow**, ***an access declaration behaves like a local declaration***. That is, it not only defines the signal access structure, but also creates the signal wire structure. That is, a shadow signal is actually bound by the first access declaration it encounters. For example, suppose we define:

```
(-device- (foo)
      (shadow X)
      (-I- foo1 Vdd X)
      (-I- foo2 X GND)
      ...)
(-device- (foo1 in out)
      (access (out (BUS 1 2)))
      (format t ''~A '' out))
(-device- (foo2 in out)
      (access (in (BUS 1 2)))
      (format t ''~A '' in))
```

The execution of

```
(make foo)
```

will print the following two lines on the screen:

```
out out
```

If `foo` reverses the order of calling `foo1` and `foo2`, then the output would be:

```
in in
```

A subtle problem arises when an unbound actual parameter meets an unbound BUS access structure (?). Since there is no way to determine the value of the question mark, WireLisp has to report the error and abort the execution. This implies that the user must re-arrange the order of device instantiations should this situation ever occur.

5.4 Signal Operations

- `(BUS-ref` *sig index*`)` or *sig*`[`*index*`]` returns a signal whose wire structure and access structure is the

index'th component of *sig*.

- `(SN-ref` *sig field*`)` or *sig.field* is similar to `BUS-ref` except access is via the given SN field.
- `(size-of` *sig*`)` returns an integer which is the number of (top-level) components in the given wire structure. For example,
  `(local (X (BUS 1 8 (SN H L))))`
  creates a signal of size 8. Be default, the size of simple signals is 1.
- `(print` *sig str*`)` prints the signal *sig* in the output stream *str*. A signal name is formed by concatenating its unique path name with its local name.
- `(-connect-` *sig1 sig2 ... sigN*`)` aliases a list of signals. That is, the physical wires represented by these signals are connected together. The output format of aliasing is determined by the user via the special function named `print-connect-simple-signal`. For example, to produce `.sim` format output, the user may define:
  ```
  (define (print-connect-simple-signal s1 s2)
          (wl-print ''= ~A ~A ~%'' s1 s2))
  ```
  Then,
  `(-connect-` *sig1 sig2*`)`
  will show up as:
  `=` *sig1 sig2*
  If the signals have hierarchical structures, the leaves in their structure trees are connected correspondingly from left to right. That implies all of the given wire structures must have the same number of leaves although they may not have the same size (the value returned by `size-of`).

There are also a number of ways of forming new signal structures from the given signals:

- `(-B-` *sig1 sig2 ... sigN*`)`: return a new signal which is the *bundling* of the given signals. The size of the new signal is N.
- `(-C-` *sig1 sig2 ... sigN*`)`: return a new signal which is the *concatenation* of the given signals. The size of the new signal is the sum of the given signals' sizes.
- `(-S-` *sig1 sig2*`)`: return a new signal which is the *top-level shuffling* of the two given signals. The size of the new signal is the sum of sizes of the given signals.
- `(-R-` *sig*`)`: return a new signal which is the *top-level reversal* of the given signal.
- `(-FL-` *sig*`)`: return a new signal which is the *bit-by-bit flattening* of the given signal *from left to right*. The size of the new signal is the same as the number of atomic wires in the given signal.
- `(-FR-` *sig*`)`: return a new signal which is the bit-by-bit flattening of the given signal *from right to left*. The size of the new signal is the same as the number of atomic wires in the given signal.

All of the above operations return an anonymous signal which has no access structure.

5.5 A Syntactic Shorthand: Iterators

Iterators[6] are used for compact representation of a list of names. An iterator is of the form:
`n1:n2`
where, `n1` and `n2` are the maximal substrings of digits appearing in a string. For example, the iterator in `data0:7` is `0:7`; and in `d2-1:4` is `1:4`. There may be an arbitrary number of iterators in a name string. For example, the iterators in `d2-1:5d2-0:1` are `1:5` and `0:1`.

[6] Iterators were suggested many years ago by Ivan Sutherland.

The way an iterator works is to produce a list of different names which vary only in the iterator portion. The default iterating step is 1 if n1 is no larger than n2. Otherwise, it is -1. For example, `foo0:5` expands to:

`foo0, foo1, foo2, foo3, foo4, foo5`

If a name has more than one iterator, it forms a nested loop with the outermost loop to the left. For example, `in0:15-0:7` expands to:

`in0-0, in0-1, ... in0-7, in1-0, in1-1, ... in15-6, in15-7`

Furthermore, the user can even specify the step of the iteration by appending string *:step* to an iterator, where *step* is a maximal substring of digits. For example, **`out0:7:2in`** expands to:

`out0in, out2in, out4in, out6in`

It is important to remember that iterators are simply syntactic shorthands. For example,

`(local in1:4)`

is equivalent to the declaration:

`(local in1 in2 in3 in4)`

which is different from:

`(local (in (BUS 1 4)))`

Iterators may appear in BUS references as well, such as `foo[1:4]`. These iterators may even contain variable bounds, such as `foo[1:N]`, where `N` is computed at run-time. Therefore, they are called ***dynamic iterators***. Iterators *not* appearing in BUS references are called *static iterators* because they are expanded into a list of names (*without parentheses*) before the program execution.

Dynamic iterators are expanded into a list of names *at run-time*. For example,

`in[1:N]`

with `N = 4` is equivalent to a list of 4 signals:

`in[1] in[2] in[3] in[4]`

So the user may use dynamic iterators in the same way as using static iterators. For example,

`(-B- in[1:N:2] in[2:N:2])`

returns a signal resulting from shuffling the wire structure of `in` so that all odd components come before all even components.

5.6 Signal Structure Types

Unlike typing in conventional languages, signal structure typing in WireLisp is more of a syntactic shorthand. The binding of structures to names via:

`(s-type` *s-name access-structure-spec*`)`

simply implies that any reference to *s-name* will be replaced *statically* by the structure it is bound to. So signal types must be defined before they are referenced.

`s-type` declarations are global, i.e. the declaration becomes effective from the point of its evaluation. If two structure types happen to have the same name, the new binding will overwrite the old one and a warning is issued. Normally, the user collects all common structure definitions in a file which is loaded before executing the program. For example, a file named **types.wl** might contain:

```
(s-type pair (SN H L))
(s-type clock (BUS 1 2 pair))
```

6 Foreign Devices

WireLisp is used to specify the *structure* of a circuit. However, there are cases where this is not a good representation. For example, programmable logic arrays (PLA) are best described by a truth table , and FSM's by a state diagram.

A device is termed *foreign* if it is defined by some means other than WireLisp drawings or procedures. Foreign devices enable the user to integrate virtually any design representation into WireLisp. However,

there must be a transformer which converts the external representation into a WireLisp procedure. For example, if the user specifies a PLA via a truth table, then there must be a program, say **pla2wl**, that compiles truth tables into WireLisp implementations. Since WireLisp has no idea what the external representation is and therefore which transformer should be used, the user must explicitly specify the conversion via the Unix **make** facility. For example, suppose the foreign device `foo` is defined via the truth table contained in the file **foo.pla**, then the Unix **makefile** must include:

```
foo.ext:     foo.pla
          pla2wl foo.pla foo.ext
```

Then the execution of the Unix command

```
make foo.ext
```

will create the WireLisp procedure for the device `foo` which is saved in **foo.ext**.

A foreign device is instantiated by executing the corresponding device procedure. A foreign device is instantiated by **-FI-** in the form:

(-FI- *dev-name* $signal_1$ $signal_2$ ... $signal_m$
(prop (*opt-param-name*$_1$ *value*$_1$) ... (*opt-param-name*$_k$ *value*$_j$)))

Since it is critical to maintain the right parameter order, WireLisp requires the user to make a drawing for each foreign device. This drawing simply defines the device header (or *device interface*) and declares access structures for the signal parameters, if necessary. The external representation must use the names as defined in the drawing. When a foreign device is loaded, WireLisp does three things:

1. *Runs the drawing through* **dp2wl** *with* **-g** *flag*. This creates the corresponding **.wl** file with the following expression automatically inserted at the end of the procedure:
   (include *device-name*.ext)
   WireLisp `include` is very similar to C **#include** (see Section 10.2)

2. *Executes the Unix command* **make** *to create the* **.ext** *file which actually defines the device.* In particular, WireLisp requires this file to be a single `block` expression.

3. *Loads the* **.wl** *file of the device.* At this point, the `include` expression is substituted by the contents of the given **.ext** file.

Foreign devices may be instantiated with general parameters as well:

(-FI- *dev-name* $signal_1$ ... $signal_m$ $gen\text{-}param_1$ ... $gen\text{-}param_n$
(gen (*gen-param-name*$_1$ *value*$_1$) ... (*gen-param-name*$_k$ *value*$_j$)))
(prop (*opt-param-name*$_1$ *value*$_1$) ... (*opt-param-name*$_k$ *value*$_j$)))

For different sets of general parameter values, there will be different pairs of **.wl** and **.ext** files generated, each defining an implementation according to the specific values. For example, suppose the foreign device `foo` has two general parameters: `W` and `D`. When `foo` is instantiated with `W=2` and `D=1`, WireLisp performs the following steps to create the device `foo_2_1`:

1. *Runs the drawing through* **dp2wl** *with* `-g ''_2_1''` *flag*. It creates the file **foo_2_1.wl** which defines the procedure `foo_2_1` with the following expression automatically inserted at the end of the procedure:
   `(include ''foo_2_1.ext'')`

2. *Executes the Unix command*:
   ```
   make W=2 D=1 foo_2_1.ext
   ```
   The makefile must have an entry such as:
   ```
   foo_$W_$D.ext:  foo.pla
         pla2wl -w $W -d $D foo.pla foo_$W_$D.ext
   ```
   Note that the macro names, such as `W` and `D` here, must be upper-case.

3. *Loads* **foo_2_1.wl**.

WireLisp has no control over what may be specified in the Unix **makefile**. This gives the user a chance to specify other related activities, such as creating *C* routines for COSMOS simulation. In fact, the user may not create the `.ext` file at all. In that case, the `include` expression is simply ignored. That is, the drawing alone defines the device.

WireLisp assumes that the root device can *never* be a foreign device.

7 Automatic Loading: `make` and `load-file`

A WireLisp program is executed by invoking `make` on the root device, e.g.

```
(make clkinv)
```

WireLisp `make` automatically loads the definition of a device when it is *first* instantiated. Similar to Unix **make**, WireLisp does not load a device if it has been loaded and its definition is still up-to-date.

If a device is instantiated by **-I-**, WireLisp assumes that it is defined either:

- By a drawing in a **.dp** file named after the device, or:
- By a procedure, in the **.wl** file named after the device.

If the **.dp** file exists, and is newer than the corresponding **.wl** WireLisp automatically invokes the schematic extractor **dp2wl** to convert the drawing into the corresponding **.wl** file. WireLisp then loads the **.wl** file of the device before executing the device procedure.

If a device is instantiated by **-FI-**, WireLisp uses the make facility to create the device file as described in the previous section.

Automatic loading ensures that every device is up-to-date and that no redundant work is performed. This implies that:

- A device is loaded only if the **.dp** or **.wl** files have changed since the device was last loaded.
- **dp2wl** is involved only if the **.dp** file is newer than the **.wl** file (or if the **.wl** file does not exist).

When `make` is invoked on a device with signal parameters, WireLisp creates the right number of *unbound* signals and passes them to the root device. These signals must be declared by access declarations which act as local declarations. For example, consider:

```
(-device- (foo a b)
        (access a (b (BUS 1 2)))
        (format T ''simple signal:  ~A ; cable component:  ~A'' a b[1]))
```

The execution of

```
(make foo)
```

will print on the screen:

```
simple signal:  A ; cable component:  B[1]
```

WireLisp fills in only the missing signal parameters. If the root device has general parameters, the user must provide their values *in the right order* when executing `make`.

The user can manually load a device as well:

(`load-file` *device-name*)

For example, `(load-file 'foo)`

The named device is loaded just like in `make`, but it is *not* instantiated.

8 Incremental Execution: `resume`

When an error occurs during execution, it is always safe to start all over again after fixing the bugs. However, some errors are continuable, i.e. the result so far is correct and the execution may resume after fixing the bugs. In these cases, it would be a great waste to start all over again, especially for very large programs.

In particular, WireLisp syntax errors are detected when a device is loaded. Since WireLisp implements dynamic loading, a syntax error may be detected long after execution starts. However, almost all syntax errors are continuable. Therefore, WireLisp provides a simple resume function — after fixing the syntax errors, the user may continue the execution by typing:

```
(resume)
```

Executing **(resume)** is equivalent to repeating the last load and then instantiating the device just loaded. This facility is *aimed only at syntax error recovery.* The user must make sure that the error is indeed continuable.

9 Output

There are two questions: What is in the output? Where is the output saved?

What is produced in the output depends on the user. Instantiating a device simply executes the corresponding device procedure. The user may insert whatever output statements are necessary in the procedure. For example, if we want to create **.sim** format output, we may define **ptrans** and **etrans** such that each prints a line describing itself: its name, connecting signals, and the width and length (see Appendix A). Instantiating a device would cause a cascade of instantiations of lower-level devices until the primitives (**ptrans** and **etrans**) are encountered. The side-effect of instantiating the root device is to produce a list of transistor descriptions corresponding to the root device instance.

By default, WireLisp uses the CMOS library which produces **.sim** format output which is saved in the **.sim** file named after the root device. In general, the user may choose different libraries and file extensions as described in the next section.

The **.sim** format output requires the following line to be at the beginning of the file:

| **units:** *units* **tech:** *technology-name* **format:** *format-name*

The three parameters, *units*, *technology-name* and *format-name*, may be set (before the program execution) via:

(**set-units** *number*)
(**set-tech** *name*)
(**set-format** *name*)

For the CMOS technology used here, their default values are:

```
units:  100 (centimicrons)
tech:  cmos-s
format:  UCB
```

10 Device Library

A device library is a set of devices for a particular implementation technology and output representation. By default, WireLisp adopts the CMOS technology producing the **.sim** format output. This is chosen when WireLisp is invoked by:

(wirelisp) or: **(wirelisp 'cmoslib 'sim)**

However, the user may choose a different library and/or a different output file extension by:

(**wirelisp** *library-name file-ext*)

This indicates that WireLisp should load the named device library (saved in *library-name*.**wl**) and adopt the given file extension. If *file-ext* is missing, **.sim** is assumed as the extension. In the library file, the user needs to tell WireLisp which devices are defined in the library. Appendix A gives a simplified example.

In general, the following operations help the user to set up the library:

- (**add-lib** *sequence-of-device-names*): add the given devices to the library device list.
- (**del-lib** *sequence-of-device-names*): delete the given devices from the library device list.

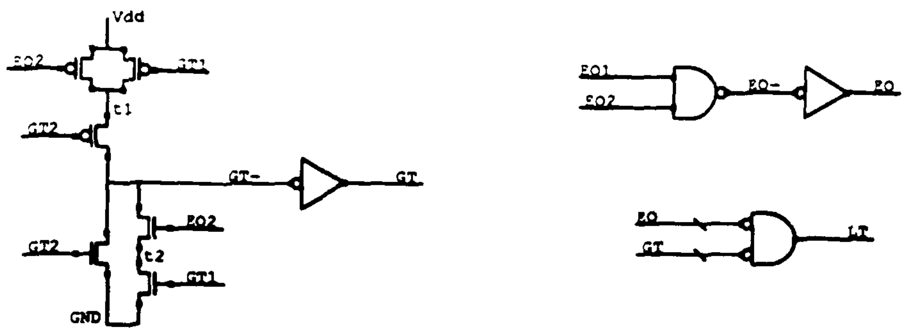


Figure 5: *logic gates computing* **GT**, **EQ**, *and* **LT**

- (`print-lib`): print out the current library device list.

In the default CMOS library, the length and width of **ptrans** and **etrans** are optional parameters with default values of 2 and 3. But in a particular design, the user may want different default values for the length and width. In that case, instead of repeatedly specifying the width and length for each transistor instantiation, the user may set them globally by:

(**set-ptrans** *length width*)
(**set-etrans** *length width*)

These new values will become the defaults.

The operations **set-ptrans** and **set-etrans** must be evaluated before program execution.

11 Examples of WireLisp Extensions

WireLisp is an open system. Just like any Lisp system, it is easy to add new features. Here are a few examples.

11.1 Equations

While much of the description of a digital system is structural, there are parts of the system which have no obvious structure, but whose input-output behavior is well-understood. As our initial attempt to incorporate behavioral description in WireLisp, **logic** expressions are provided for the inclusion of simple logic equations as replacements for small collections of logic gates. For example, the logic gates computing **GT**, **EQ**, and **LT** in Figure 5 could be replaced by the following three expressions:

```
(logic GT (or GT2 (and EQ2 GT1)))
(logic EQ (and EQ1 EQ2))
(logic LT (and (not EQ) (not GT)))
```

Each logic expression is equivalent to defining a device whose behavior is described by the equation and instantiating that device in the context of the logic expression. A set of Lisp functions have been written for translating logic expressions directly into the equivalent WireLisp device procedures.

11.2 include

WireLisp **include** expressions are similar to C **#include** directives:

(**include** *file-name*)

The expression is replaced by the contents of the given file. If the file does not exist, this expression is simply ignored.

11.3 Functions for COSMOS Simulations

There have been a number of functions added for COSMOS simulations.

1. (**cos-func** *dev-name inst-name* (*input-signals*) (*zero-delay-signals*) (*output-signals*))

   This function prints out one line in the output file, which looks like:

   B *extended-inst-name dev-name input-signals* ; *zero-delay-signals* ; *output-signals*

   Here, *extended-inst-name*, which is the concatenation of the invoking device instance name and the given *inst-name*, indicates the *path of this particular instantiation*. *dev-name* gives the name of the functional block called. Signals within a group are separated by blanks. Simple signals are printed as they are, and cables are flattened with the wire structure leaves enumerated *from right to left* because BUS indices are assumed to range from least to most significant bits.

2. (**inputs** *input-signals*)

   This function prints out one line for each given signal:

   A *signal* **Sim:In**

   If the input signal is a cable, it is flattened with the wire structure leaves enumerated *from left to right*. One leaf occupies one line.

3. (**outputs** *output-signals*)

   This function prints out one line for each given signal:

   A *signal* **Sim:Out**

   Cables are treated in the same way as in **inputs**.

4. (**vectors** *cable-signals*)

   This function prints out one line for each given cable:

   V *cable-name sequence-of-cable-components*

   Here, cable components refer to the leaves of the cable wire structure which are enumerated *from right to left*. Simple signals *cannot* appear in a **vector** expression.

A An Example of Device Library

```
;;; define how to print aliasing signals.
(define (print-connect-simple-signal s1 s2) ;produce .sim output
     (wl-print ''= Ā Ā %'' s1 s2))
;;; define what is in the library.
(add-lib etrans ptrans)
(-device- (etrans gate source drain (L 2) (W 3)) ;optional length & width
     (wl-print ''e Ā Ā Ā %'' gate source drain L W))
(-device- (ptrans gate source drain (L 2) (W 3))
     (wl-print ''p Ā Ā Ā %'' gate source drain L W))
```

B Signal Access Structure Syntax

Note that: (1) Curly brackets represent the iteration of one or more times. (2) Any name which is legal in T is also legal in WireLisp. (3)*null* means nothing should be there.

⟨*local-signal-decl*⟩:: (**local** {⟨*field*⟩})

⟨*access-decl*⟩:: (**access** {⟨*field*⟩})

⟨*field*⟩:: ⟨*name*⟩| ({⟨*name*⟩} ⟨*struct-spec*⟩)

⟨*struct-spec*⟩:: ⟨*SN-spec*⟩| ⟨*BUS-spec*⟩| *null*

⟨*SN-spec*⟩:: (SN {⟨*field*⟩})

⟨*BUS-spec*⟩:: (BUS ⟨*low*⟩ ⟨*high*⟩ ⟨*struct-spec*⟩)

where, both *low* and *high* can be any integer expression. The index advance step is 1 or -1, depending on if *low* is smaller or larger than *high*.

ACKNOWLEDGEMENT

We are very grateful to Robert Condon for his patience in experimenting WireLisp and for his many good suggestions; and Larry McMurchie for his help in preparing this manual; and Martine Schlag for making T available and for her help in the initial implementation. Finally, thanks to all who have helped in the development of WireLisp.